

## Overview of Managed PostgreSQL with Amazon RDS and Aurora

Information from AWS training and certification, Amazon Aurora PostgreSQL, and Amazon RDS PostgreSQL, summarizing key points of what I've learned

### ***Relational Databases:***

Relational database is a type of database that stores and provides access to data points that are related to one another. They are more intuitive, and straightforward by representing data in tables utilizing relational model.

Each row in a table is a record with a unique identifier called a *key*, and the columns of the table hold attributes of the data. Each record usually has a value for each attribute establishing the relationships among data points. Foreign keys are used to link tables to one another.

Relational database model provides a standard way of representing and querying data. Something that makes relational database stand out is in the use of tables and indexes to better structure information conveniently.

### ***Amazon RDS:***

Relational databases are widely used from personal projects to a large-scale application in corporate setting. However, relational databases can be hard to manage as you make updates or scale. Amazon Relational Database management, RDS, helps you and your team to manage the relational database of your choice in a handy manner.

**Multi Engine Support:** Amazon RDS gives you access to the capabilities of a familiar database. It supports MySQL, MariaDB, PostgreSQL, Oracle Server, and Microsoft SQL Server.

**Automated Tasks:** Amazon RDS manages the work involved in setting up a relational database, from provisioning the infrastructure capacity you request to installing the database software. After your database is set up, Amazon RDS automates common administrative tasks such as performing backups and patching the software that powers your database. Amazon RDS also automates scaling, replicas, and restore actions.

**Scalability to handle growth:** You benefit from the flexibility of being able to quickly scale the compute resources or storage capacity associated with your relational DB instance. Amazon RDS uses replication to enhance database availability, improve data durability, or scale beyond the capacity constraints of a single DB instance for read-heavy database workloads.

**Multi-AZ Deployment:** Amazon RDS Multi-AZ deployments provide enhanced availability and durability for RDS DB instances, making them a natural fit for production database workloads.

### ***Amazon RDS Feature Highlights:***

Amazon RDS Multi-AZ Deployments: Amazon RDS Multi-AZ deployments provide enhanced availability and durability for RDS DB instances. When you provision a Multi-AZ DB instance, Amazon RDS automatically creates a primary DB, which then synchronously replicates the data to a standby instance in a different availability zone. Each availability zone runs on its own physically distinct independent infrastructure and is engineered to be highly reliable.

### ***Amazon RDS Performance Insights:***

Database administrators need to monitor and manage their databases, but the Amazon RDS performance insights feature can help you quickly assess any performance bottlenecks in your relational database workloads. Performance insights collect detailed database performance data and display the data to drive a graphical interface.

### ***Aurora:***

Aurora is a cloud-based relational database management that is compatible with MySQL and PostgreSQL. It offers speed that is five times faster than standard MySQL and three times faster than PostgreSQL. Aurora offers fault-tolerant, self-healing storage which provides six copies of data across three Availability Zones and continuously generate backup to Amazon Simple Storage Service.

Aurora is highly secure and offers network isolation and encryption at rest and in transit. And furthermore, it has the same management benefits as Amazon RDS, meaning no hardware provisioning, software patching, setup, configuration, nor backups.

### ***Installing and initiating AWS RDS Postgres from scratch:***

1. First, user must have access to AWS and have a valid account (Can be free)
2. Navigate to Amazon RDS
3. Select the region for the DB instance
4. Click “Create Database” button
5. Choose your engine option as “PostgreSQL”
6. Choose the version of the engine
7. Choose “Free tier”
8. Use the following settings for your DB instance as an example:

**DB instance identifier:** Enter a name for the DB instance that is unique for your account in the Region you selected. For this example, we will name it *rds-postgresql-sample*.

**Primary username:** Enter a username that you will use to log in to your DB instance. We will use *PrimaryUsername* in this example.

**Primary password:** Enter a password that contains 8–41 printable ASCII characters (excluding /, ", and @).

**Confirm password:** Retype your password.

9. When you follow the above steps, you should be seeing the following screen:

**Set**

**DB instance identifier** [information](#)  
Enter the DB instance name. The name must be unique for all DB instances owned by an AWS account in the current AWS Region.

DB instance identifiers are not case-sensitive, but are stored in all lowercase, such as 'mydbinstance'. Constraints: Must consist of 1-60 alphanumeric characters or hyphens. The first character must be a letter. Two hyphens cannot be consecutive. It cannot end in a hyphen.

**▼ Set Credentials**

**Master username** [information](#)  
Enter the login ID for the DB cluster's master user.

1 to 16 alphanumeric characters. The first character must be a letter.


☐ **Automatic password generation**  
Amazon RDS can generate a password for you, or you can specify your own password.

**Master password** [information](#)


Constraints: At least 8 printable ASCII characters. Cannot contain: / (slash), ' (single quote), " (double quote), and @ (at sign).

**password confirmation** [information](#)

10. Set DB instance size to: db.t2.micro-1 vCPU, 1Gib RAM
11. Storage settings: Set your storage type to General Purpose (SSD), Allocated Storage to 20Gib, and enable storage auto scaling with multi-az deployment turned off.
12. Connectivity: Set your network and security settings to: VPC to default VPC, subnet group to default, public access to yes, VPC security group to create new, and availability zone to no preference.
13. Authentication: In this practice scenario, choose password authentication.
14. Lastly, under “Additional Configuration”, set your database port to 5432.
15. For the database options, enter your initial database name and db parameter group
16. Set your backup settings to: Enable automatic backups, backup retention period to 1 day, and backup window to “No Preference”, and copy tags to selected
17. Click “Create Database”
18. And you should be able to create your own DB instance and redirected to the screen below:

 **Creation of the rds-postgresql-sample database was successful.**

[View connection details](#)



### ***Prompts – PostgreSQL:***

Prompt characters can be used to return types of information in psql, and used to short-handily process commands without typing the command in full.

Prompt	Description
%M	Full host name of the database server
%m	Host name of the database server
%o>	Port number
%n	Database session user name
%/	Name of the current database
%p	Process ID of the current backend
%x	Transaction status
%l	Line number inside the current statement

### ***Watch Queries:***

Watch queries were used in need to view all the currently running queries and can be done by setting up /watch command. It can be set so that watch queries are ran in time intervals and can be included in a script

### ***Data Definition Language and Data Manipulation Language:***

You can use numerous commands to create and modify a database in PostgreSQL, and these commands are categorized under two different universal languages: Data Definition Language (DDL) and Data Manipulation Language (DML)

DDL: DDL is a set of commands to help you perform CRUD on database. Some of the commonly used DDL commands are: CREATE TABLE, DROP TABLE, CREATE SEQUENCE, DROP SEQUENCE

DML: DML is a set of commands that helps you retrieve, store, change and delete data in your database. Some common DML commands are: SELECT, INSERT, UPDATE, DELETE

### ***SQL Functions:***

SQL functions are database objects that are commonly used for processing or manipulating data. When a function is used as shown in this example, the user is formatting how the data is displayed. Think of any sort of database or even Microsoft Excel, in which built-in functions help you process or modify information.

### ***Nested Statements:***

PostgreSQL is very flexible in a sense user may insert nested statement almost everywhere. User may nest a PostgreSQL query inside statements such as SELECT, INSERT, UPDATE, and DELETE and tie multiple statements, joining data together across different tables. Nested statements are also called subqueries.

### ***Joins:***

In a relational database, data is distributed in multiple logical tables. To get a complete, meaningful set of data, you need to query data from these tables by using joins. Each join type specifies how the data from one table will be used to select rows in another table.

There are multiple types of joins including inner join, left outer join, right outer join, and full outer join which can be used to specify the data user may want depending on the situation.

Alias is very helpful when using joins to simplify the table names and column names in a complicated query. Aliases allow you to use a shorthand name within the query helping you to be a more efficient DML.

As an example, table name insurance can be aliased as i, and subsequently its column name can be aliased as c. Which clears up a lot of mess when joining multiple tables by typing i.c instead of insurance.column\_name.

### ***Integrity Constraints:***

When designing tables, you might want to constrain data from individual columns and the tables themselves so that business rules can be enforced. This can be done using integrity constraints. In modern database best practice, constraints are not used in the database object. Note that PostgreSQL will allow the use of integrity constraints if the user chooses to use them.

Integrity constraints help ensure that values in one table make sense with related data in another table. Commonly used constraints include NOT NULL, CHECK, UNIQUE, PRIMARY KEY, and FOREIGN KEY.

### ***Combining Queries:***

There are times when you want to compare query results where best practice advises not to use a join statement. This is because data is being pulled from different result sets of those queries and combined into a single result.

When you want to combine queries, you can use UNION, INTERSECT, and EXCEPT. You can use these clauses to combine or exclude like rows from two or more tables. They are useful when you need to combine the results from separate queries into a single result. They differ from a join in that entire rows are matched. As a result, they are included or excluded from the combined result.

### ***Aggregates:***

In PostgreSQL, aggregate functions can be used to compute a single result from multiple input rows. This can be used to help the user to either target the needed information you need to display information or to eliminate certain information from display.

Some of the common aggregate functions include: AVG(), MIN(), MAX(), and SUM(). However, GROUP BY clause itself can be used to aggregate a set of rows to group the outcome of the query based on previously mentioned functions.

### ***Copy Command:***

The copy command is an interesting command as it is used to directly move data between tables and system files. You can specify whether data is coming to or going from the server by adding TO or FROM clause to the query and it is a server-side command. The command requires user to specify the viewpoint of the server, and the user ID logged in to the server must be able to access the file.

Copy, or \copy command runs as a SQL COPY command, therefore can be used interchangeably, however \copy command needs to be processed through psql; reading or writing files and routes of the data between the server and the local system.

\copy from and \copy to commands import and export data to or from a database, better described as pull and pushing the data from a source to your database. These commands can be used concurrently with supplementary commands like REPLACE, APPEND, and RULE.

*STDIN and STDOUT formatting*

```

\copy table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]

\copy { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]

```

### *Copy Options List*

Option	Description
DELIMITER	Character that separates columns
NULL	String for a null value
HEADER	If the file contains a header line
QUOTE	Quoting character to be used
ESCAPE	Escape character

### *Foreign Data Wrappers*

A PostgreSQL FDW is an installed extension that creates a link to another PostgreSQL database. It can move data between databases. The postgres\_fdw module provides the FDW postgres\_fdw, which can access data stored in external PostgreSQL servers.

When you are linked to the server using the FDW with the qualified name postgres server, you can run other commands and group, or aggregate, them all together in a command string.

```

CREATE SCHEMA dw;

CREATE FOREIGN TABLE dw.foo_stat (
    coll_sum int,
    coll_avg float
) SERVER postgres_server
OPTIONS (table_name 'foo_stat', schema_name 'public');

INSERT INTO dw.foo_stat
SELECT sum(coll), avg(coll)
FROM foo;

```

For users to successfully make full use of PostgreSQL FDW, user should:

- 1) user should first install the extension by using the CREATE EXTENSION command
- 2) Create a foreign server object by using CREATE SERVER, specify connection information and options
- 3) Create user mapping by using CREATE USER MAPPING, and this will allow access to each foreign server for each database user. In this command, you should specify remote username, password, and password options for the user mapping
- 4) Create foreign table is the next step, using CREATE FOREIGN TABLE or IMPORT FOREIGN SCHEMA for each remote table you want to access. The columns of the foreign table must match the referenced remote table, and there by specify correct remote names as options of the foreign table object

### ***Pgloader Utility***

The pgloader utility is a data loading tool that is based off on PostgreSQL COPY protocol to import the data into the server. It manages errors by filling a pair of reject.dat and reject.log files.

PGloader offers several advantages over using the COPY command which will be listed below:

- 1) Supports more file formats, such as: CSV, Fixed-column formats, DB files, and IBM IXF files
- 2) Skips bad records when performing a copy of the data
- 3) Parallelism supported, which allows loading of more than one file at a time
- 4) Available in supported PostgreSQL repositories ex. yum, apt

In the context of pgloader, a command file (or load file) is a file that instructs pgloader how to perform a migration. Using a command file, you can list all pgloader commands you want to run and save them to a single file. You can then run that file and several commands together for greater efficiency. This gives you finer control over how your data is loaded into PostgreSQL and helps you to perform complex migrations.

The command copies information from a CSV format command file into specific columns of a table in PostgreSQL. pgloader uses a command file, or you can run it all from the command line.

### ***Using the pgloader from command file***

- 1) Load the command file from the CSV file. CSV file, user credentials, and SSL required
- 2) The ~/ UNIX command instructs pgloader where to find that CSV file
- 3) The connection information to the target database is provided.
- 4) Target table and columns need to be specified



```

LOAD CSV
  FROM '-/us_cities_states_counties.csv'
  INTO
postgresql://user:password@pg11.rds.amazonaws.com/imdb?sslmode=require
  TARGET TABLE cities
  TARGET COLUMNS
  (
    name,
    state,
    state_name,
    county,
    alias
  )

```

- 5) Set field parameters
- 6) Set the standard\_conforming\_strings property to 'on'
- 7) Allocate 12 megabytes of working memory per set

```

WITH truncate,
  skip header = 1,
  fields optionally enclosed by '"',
  fields escaped by double-quote,
  fields terminated by '|'

SET standard_conforming_strings to 'on',
  work_mem to '12MB'

```

- 8) Check for existence of a table
- 9) Creates the table with the specified parameters, if it does not exist
- 10) Analyzes the table contents

```

BEFORE LOAD DO
  $$ CREATE TABLE IF NOT EXISTS cities (
    id serial PRIMARY KEY,
    name varchar,
    state varchar,
    state_name varchar,
    county varchar,
    alias varchar);
  $$;

AFTER LOAD DO
  $$ ANALYZE cities; $$;

```

## ***Security Management***

Security for RDS and Aurora PostgreSQL can be managed at three levels:

- 1) AWS Identity and Access Management (IAM)
- 2) Amazon Virtual Private Cloud (Amazon VPC)
- 3) Standard PostgreSQL security management

To get more into details for these three levels:

**IAM:** Controls who can perform management actions on Aurora DB clusters or RDS DB instances, user is recommended to use IAM. AWS account must have IAM policies that grant the permissions required to perform RDS management operations.

**Amazon VPC:** Aurora DB clusters must be created using Amazon VPC, because devices and Amazon EC2 instances can open connections to the endpoint and port of the DB instance for Aurora DB clusters in VPC, you must use an AMAZON VPC security to control these connections.

**PostgreSQL:** To authenticate login and permissions of an Aurora DB cluster, you can take the same approach as with a standalone instance of PostgreSQL. Commands such as CREATE ROLE, ALTER ROLE, GRANT, and REVOKE works just as they do in on-premises databases.

## ***RDSadmin default privileges***

When each new object is created, the rdsadmin role is automatically created as a security measure. DBA is assigned the role of rdsadmin. DBAs have default privileges to manage the database and other users.

RDS DBA has the following default privileges when DB instance is created

- LOGIN
- RDS SUPERUSER
- INHERIT
- CREATEDB
- CREATEROLE
- VALID UNTIL 'infinity'

Rdsadmin user is created upon db cluster creation to provide service management for Aurora DB cluster.

## ***Object Security Auditing***

Auditing is a method of verifying that everything is working as expected without any unauthorized access. Users can audit databases, roles, tables, or columns and can be done through

pgaudit extension. This extension offers DBA the ability to take control of the object audit inside the database. Object security involves tracking who can access data in addition to restricting access.

First, DBA must create a specific rds\_pgaudit role. And then the DBA needs to modify the DB parameter group associated with the instance and set the role of pgaudit to the newly created rds\_pgaudit. Afterwards, all the users need to do is to reboot the instance.

The audit log can include the following classes: READ, WRITE, FUNCTION, ROLE, DDL, AND MISC.

Classes	Description
READ	SELECT and COPY FROM
WRITE	Examples: INSERT, UPDATE, DELETE, TRUNCATE
FUNCTION	Function calls and DO blocks
ROLE	Statements related to roles and privileges
DDL	All data description language (DDL) that is not included in the ROLE class
MISC	Miscellaneous commands

### ***Object Auditing***

When the audit log is enabled, it will audit everything in the database. But DBA might not need to audit anything. If data is to come from an application server that have passed through quality assurance and be vetted, audits mainly need to occur on anything coming from a user into the database.

This is conducted by setting different object-level audits on specific tables. As an example, DBA might audit if a user input or viewing information on a table. They are not required to audit information in a table that includes information that is irrelevant. In other words, DBA can create object-level audits on sensitive or confidential information but select not to audit public information.

### ***Reading the Audit Logs***

Audit records are added to the normal PostgreSQL logs, they often come in as a string and displayed in a CSV file. Below image includes the field seen in the log with a description of the types of information it contains.

Field	Description
AUDIT_TYPE	SESSION or OBJECT
STATEMENT_ID	Unique statement ID for the session
SUBSTATEMENT_ID	Substatement ID
CLASS	Examples: (READ, ROLE)
COMMAND	Examples: ALTER TABLE, SELECT
OBJECT_TYPE	Examples: TABLE, INDEX, VIEW
OBJECT_NAME	The fully qualified object name
STATEMENT	The statement runs on the backend
PARAMETER	Statement parameters

### ***Backups***

PostgreSQL backups can be either physical or logical. Physical backups are best known as file-system-level snapshots while logical backups are best known as SQL dumps. Both backup types come in handy, however has it's pros and cons.

**Physical backup:** An on-premises physical backup is done by copying individual directories and files, and then restoring them one at a time. For Amazon RDS and Aurora, the automated backup uses the physical backup mode but uses point-in-time restore (PITR) to back up and recover the entire database.

**Logical backup:** Logical backups use multiversion concurrency control (MVCC) to generate a consistent, logical backup of data from inside the database. Logical backup, or SQL dump, works by generating a text file with SQL commands to recreate the PostgreSQL cluster, a database, or a given table.

PostgreSQL offers two main backup approaches to address the different needs of its users.

## ***Automated Monitoring Tools – Scrapped from AWS skill builder***

AWS provides tools to monitor Amazon RDS and Aurora. DBAs can configure some of these tools to automate monitoring. Some other tools may require DBAs to manually intervene. What is recommended is to automate monitoring tasks as much as possible.

**Amazon RDS events:** Amazon RDS events subscribes users to receive notifications when changes occur with a DB instance, DB cluster, DB cluster snapshot, DB parameter group, or DB security group.

**Database log files:** Database log files enable you to view, download, or monitor database log files using the Amazon RDS console or Amazon RDS application programming interface (API) operations. You can also query some database log files that are loaded into database tables.

**Amazon RDS Enhanced Monitoring:** Amazon RDS Enhanced Monitoring provides metrics in real time for the operating system (OS). This is similar to how a user can use Secure Shell (SSH) to access information in a database server.

**Amazon RDS Performance Insights:** Performance Insights assesses the load on your database and determines when and where to act.

**Amazon RDS recommendations:** Amazon RDS recommendations look at automated recommendations for database resources, such as DB instances, DB clusters, and DB cluster parameter groups.

Enhanced Monitoring using Amazon RDS and Aurora monitors two things

- 1) OS metrics including the central processing unit (CPU), memory, and I/O.
- 2) Database Performance

## ***Monitoring Logs***

DBAs are given options to automatically generate error logs through configuring Amazon RDS. The logs will load via Amazon CloudWatch. Once error logs are placed in CloudWatch, user may set alerts based on specific errors.

The screenshot displays the AWS CloudWatch console interface. The left-hand navigation pane includes links to CloudWatch, Dashboards, Alarms, Billing, Events, Rules, Event Buses, Logs, Insights, Metrics, and Alpine. The 'Logs' section is currently selected. The main content area shows the breadcrumb path: CloudWatch > Log Groups > /aws/rds/instance/pg11-test/postgresql > All streams. Below this, there are controls for 'Expand all' (set to 'Row'), 'Text', and a refresh icon. A filter bar is set to 'ERROR' with a dropdown menu showing 'all', '30s', '5m', '1h', '6h', '1d', '1w', and 'custom'. A table of log entries is displayed with columns for 'Time (UTC +00:00)', 'Message', and 'Show in stream'. The logs are grouped by date: 2019-05-03, 2019-05-06, and 2019-05-09. Each entry includes a timestamp, a detailed message with IP addresses and port numbers, and a link to view the log in the stream.

Time (UTC +00:00)	Message	Show in stream
2019-05-03		
18:52:26	2019-05-03 18:52:26 UTC: 172.31.33.158(36150) postgresql	<a href="#">pg11-test.1</a>
18:53:09	2019-05-03 18:53:09 UTC: 172.31.33.158(36150) postgresql	<a href="#">pg11-test.1</a>
19:22:43	2019-05-03 19:22:43 UTC: 172.31.33.158(36192) postgresql	<a href="#">pg11-test.1</a>
2019-05-06		
14:40:17	2019-05-06 14:40:17 UTC: 172.31.33.158(36662) postgresql	<a href="#">pg11-test.0</a>
14:40:38	2019-05-06 14:40:38 UTC: 172.31.33.158(36662) postgresql	<a href="#">pg11-test.1</a>
2019-05-09		
13:19:33	2019-05-09 13:19:33 UTC: 172.31.33.158(37152) postgresql	<a href="#">pg11-test.1</a>
13:21:05	2019-05-09 13:21:05 UTC: 172.31.33.158(37154) postgresql	<a href="#">pg11-test.0</a>
13:24:22	2019-05-09 13:24:22 UTC: 172.31.33.158(37156) postgresql	<a href="#">pg11-test.0</a>
19:07:52	2019-05-09 19:07:52 UTC: 172.31.33.158(37200) postgresql	<a href="#">pg11-test.1</a>
19:09:20	2019-05-09 19:09:20 UTC: 172.31.33.158(37202) postgresql	<a href="#">pg11-test.0</a>
19:23:16	2019-05-09 19:23:16 UTC: 172.31.33.158(37208) postgresql	<a href="#">pg11-test.0</a>
19:26:00	2019-05-09 19:26:00 UTC: 172.31.33.158(37210) postgresql	<a href="#">pg11-test.1</a>
19:27:10	2019-05-09 19:27:10 UTC: 172.31.33.158(37214) postgresql	<a href="#">pg11-test.0</a>

## Wait Events

The wait\_event\_type column indicates the type of events are a great indicator for the types of events the backend is waiting on. NULL indicates that there is no event specified. There always are wait event types and wait event names, for example, there might be an LWLock and wait events may have 1 to up to 65 wait event names.

The wait event name is a great indicator for what exactly is being waited on. Often times user can get hints from the name of the wait event. For example, ShmemIndexLock is a wait event for backend waiting to find or allocate space in shared memory.

Two types of wait events as an example: One instance of Lock and seven of LWLockNamed. The wait\_event named types are WALWriteLock and transactionid.

```
SELECT pid, wait_event_type, wait_event
FROM pg_stat_activity WHERE wait_event is NOT NULL;
```

pid	wait_event_type	wait_event
1053	LWLockNamed	WALWriteLock
1054	LWLockNamed	WALWriteLock
1055	LWLockNamed	WALWriteLock
1056	LWLockNamed	WALWriteLock
1057	LWLockNamed	WALWriteLock
1059	LWLockNamed	WALWriteLock
1061	Lock	transactionid
1062	LWLockNamed	WALWriteLock

(8 rows)

## Wait Event Types

There are nine different wait event types, and following table below shows the type name and description of the event types.

Type	Description
LWLock	Lock protecting a data structure in shared memory
Lock	Lock protecting SQL-visible objects such as tables
BufferPin	Waiting for access to a data buffer
Activity	Waiting for system processes
Extension	Waiting for activity in an extension module
Client	Waiting for some activity on a socket
IPC	Waiting for another process in the server
Timeout	Waiting for a timeout to expire
IO	Waiting for an IO to complete

## *PostgreSQL System Catalog*

The PostgreSQL system catalog can be described as a schema with tables and views and contain metadata about all objects in the database. DBAs can utilize this in many ways including specifying operations that are occurring, access records of tables and records, and database functionalities such as reading information from memory or disk.

In addition to public and custom schemas, all databases should contain pg\_catalog schema. This schema tend to contain the system tables and all of the built in data types, functions, and operators. Pg\_catalog acts just like a dictionary for data for PostgreSQL ran databases.

There are a total of 60 catalogs, which can be distinguished in 3 categories:

**Structural:** Used to manage the relational management system (RDBMS)

**Informational:** Used to view table size and monitor activity

**Performance:** Used to monitor performance statistics about queries, tables, and indexes

## *Vacuum Command*

PostgreSQL databases require periodic vacuuming. This prevents XID wraparound from causing problems and potential exhaustion. User will be given a choice to run vacuuming manually or automatically and details of the vacuuming will be discussed below.

Manual Vacuum mode:

There are two variants of VACUUM: standard VACUUM and VACUUM FULL. We will cover the standard VACUUM first. The standard default VACUUM command vacuums the entire table and all associated indexes. It looks for free space and marks it in the Free Space Map (FSM).

The FSM keeps track of pages that have free space available for use. You can add parameters to the VACUUM command to fine-tune results and accomplish additional tasks.

Options for manual vacuum mode:

VACUUM [table]: This is the basic command used to vacuum a table

VACUUM ANALYZE [table]: This command vacuums and performs a statistical analysis

VACUUM ANALYZE FREEZE [table]: This command vacuums, performs a statistical analysis, initiates the FREEZE operation, and resets xmin (the identity of the inserting transaction for this row version) to 2.

VACUUM ANALYZE VERBOSE [table]: This command vacuums, performs a statistical analysis, and prints the statistics

### ***Automatic Vacuum mode:***

PostgreSQL has an optional but highly recommended feature called autovacuum. Autovacuum is a daemon that automates the launch of VACUUM and ANALYZE commands (to gather statistics). Autovacuum checks for bloated tables in the database and reclaims the space for reuse. The autovacuum daemon is activated by default in Aurora and Amazon RDS PostgreSQL.

In the default configuration, autovacuuming is activated and the related configuration parameters are appropriately set. VACUUM can run manually. However, PostgreSQL recommends that you run the autovacuum daemon.

The autovacuum daemon consists of multiple processes. There is a persistent daemon process, called the autovacuum launcher, that oversees starting autovacuum worker processes for all databases.

There is a maximum of *autovacuum\_max\_workers* worker processes that can run at the same time. If there are more processes than the maximum, the next database will process as soon as the first worker finishes.

Autovacuum follows these steps on execution:

- 1) Wake up
- 2) Look for a table that has hit a certain threshold
- 3) Vacuum the table
- 4) Sleep

### ***Database Scaling:***

All databases are bound to use up resources such as CPUS, memory, and disk I/O to run queries and commands. If your database's workload is to increase without scaling up the resources required, database performance will be negatively impacted.

There are three ways to improve the scalability for your database, and these improving methods are known as

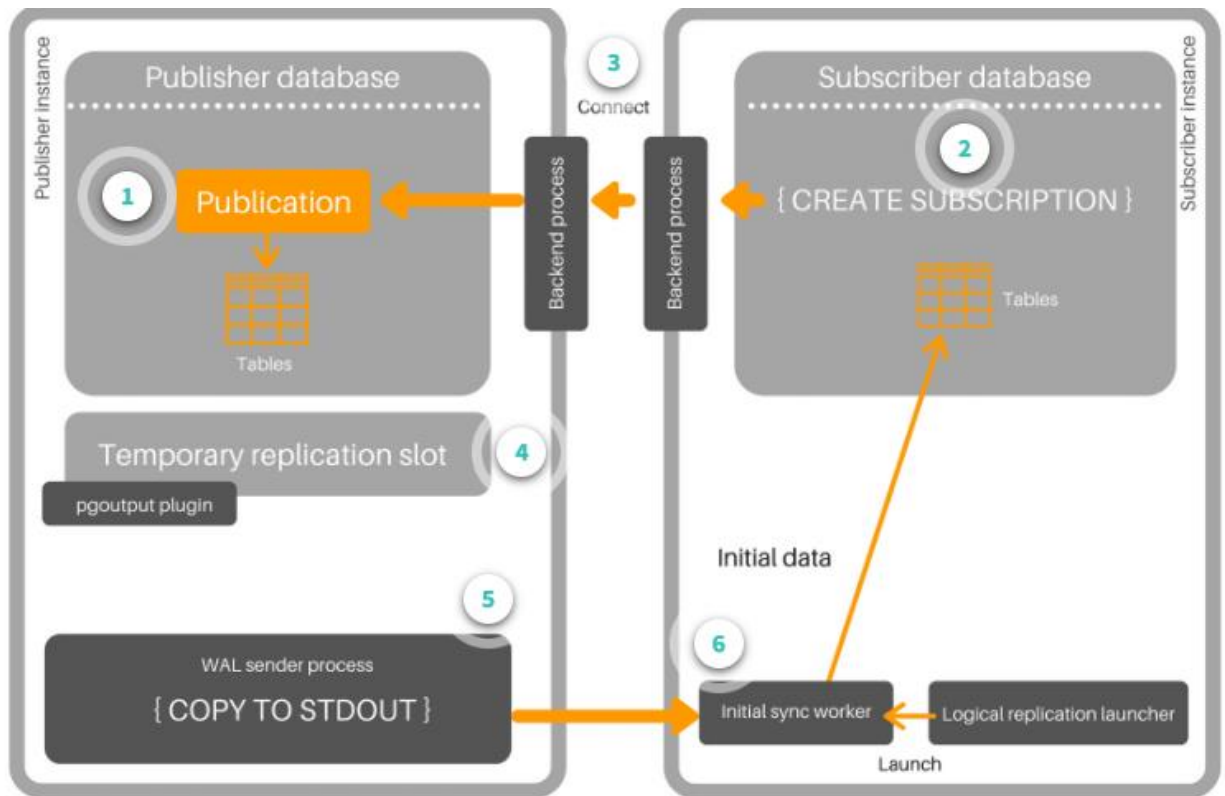
- 1) Optimizing: Optimizing uses database resources more efficiently through performance tuning and connection management
- 2) Vertical Scaling: Vertical scaling identifies the increasing requirements for resources for performance and provides more resources to your DB instance.
- 3) Horizontal Scaling: Horizontal scaling helps you process more queries using additional DB instances, reporting and analytic applications create a heavy workload. This method offers effective and sound solution



## Logical Replication

Users may build logical replication supplementary to the streaming replication by applying a logical decoding process. This process makes use of publish-and-subscribe model, and it utilizes subscribers to pull data from the origin of the data and starts copying a snapshot of the data.

When the process is complete, the changes made on the publisher are handed over to the subscriber and the subscriber may apply data in the same order as they applied commits on the publisher, helping with the transactional consistency.

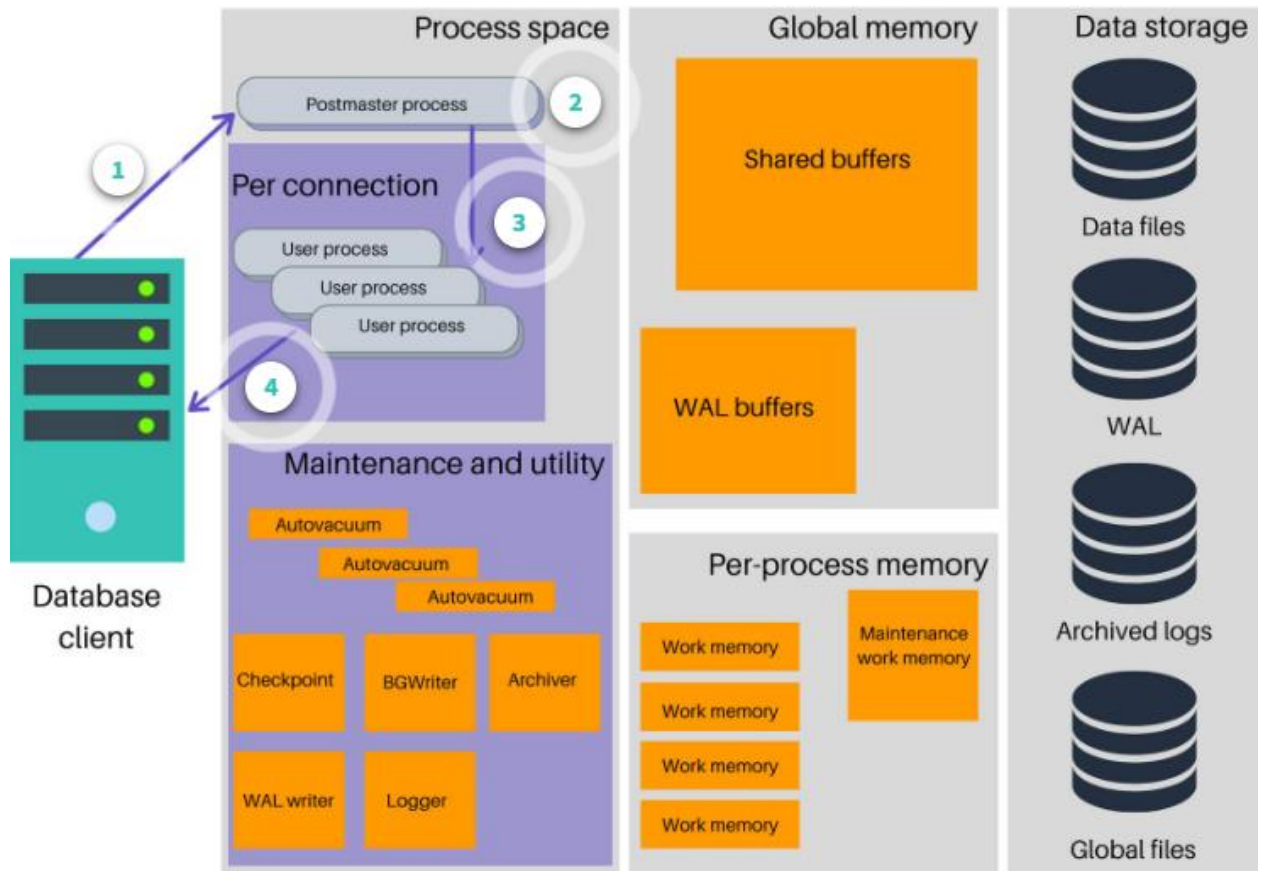


## Managing Database Connections

Database connections are the most fundamental part of working with databases, and yet it may come difficult to some. Database connections in PostgreSQL is expensive and use up a lot of resources. However, PostgreSQL has a built-in functionality for reusable connections users can use to reduce these problems.

PostgreSQL creates new child process for every database session. These are backend process for the user and it helps avoiding database overloading and managing these connections for applications effective for requests and processes.

Resources are used upon opening up a new database, and this is inevitable. Resources must perform several steps to create a new connection, and thereby increased resource overhead when new connect requests are made. Below image well demonstrates information related to the connection process



### ***Database Proxies***

Adding a database proxy decreases the time needed to execute commands and decrease connection overhead. By implementing proxies, user may expect to reuse existing or duplicate connections. This can help user direct writes to the primary while offloading read requests to the read replicas.

Proxies can be handy when there are many application servers or rely on serverless applications to create connections shared by all your other applications.

### ***AWS DMS Console***

There are many types of source database, but these are the top source database that best represents source database

- 1) Source database that is located on user's premises outside of AWS
- 2) Source database that is running on Amazon EC2 instance
- 3) Source database that is an Amazon RDS database

### ***Creating a replication instance***

First step to taking when migrating a database is to create a replication instance (obviously). AWS DMW replication instance is a managed Amazon EC2 instance that is capable of hosting one or more replication tasks. This instance is capable and offers sufficient storage room and power to process and perform the tasks you input to migrate data from your source database to the target database.

To begin, user should:

- 1) Log in to AWS and navigate to the AWS DMS console for your region, and choose "Create Replication Instance"
- 2) OR you can navigate to "Replication Instances" and choose Replication instances > Create replication instance

After creating a replication instance, user should specify replication instance information to further progress. Here are some requirements for the specification:

Name: This field may contain up to 63 ASCII characters excluding /, ", and @

Description: A brief description of your replication instance excluding \_:/=+@ with maximum of 1000 characters

Instance class: Instance class with the configuration. Dictates the storage amount, network, and processing power for user migration

Engine Version: Engine version is set to the latest version of the AWS DMS replication by default. However, user may set the engine version to the previous versions if wanted.

Allocated storage (Gib): Storage is primarily consumed by log files and cached transactions. For AWS DMS to successfully stream cached changes, replication instances has to have sufficient memory.

VPC: Virtual private cloud (VPC) is essential to locate your source or target database. Replication instance must be able to access the data in the source VPC.

Multi-AZ: Optional parameter. It is used to create a standby replica of your replication instance.

Publicly accessible: This option is mandatory if you want the replication instance to be accessible from the internet



## Certificate of Completion

**Jay Lee**

successfully completed

**Amazon Aurora PostgreSQL and Amazon RDS PostgreSQL**

on

12/10/2022